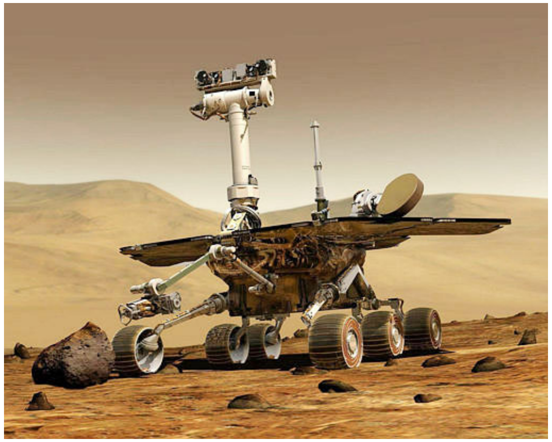
Combining Model Checking and Symbolic Execution for Software Testing

Corina Păsăreanu

Carnegie Mellon Silicon Valley, NASA Ames

software is everywhere



errors are expensive …

annual cost of software errors to US economy is $ ~60B [NIST'02]

main approaches to finding errors

- model checking
  - automatic, exhaustive
  - scalability issues; reported errors may be spurious
- static analysis
  - automatic, scalable, exhaustive
  - reported errors may be spurious
- testing
  - reported errors are real
  - may miss errors
  - well accepted technique; state of practice

our approach

combine model checking and symbolic execution for test case generation

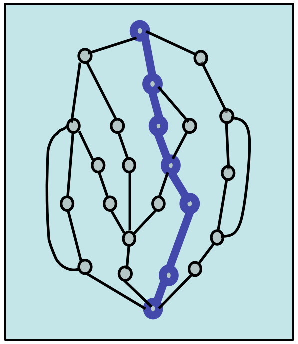
Model Checking vs Testing

program / model

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

test oracle

testing / simulation



OK

error

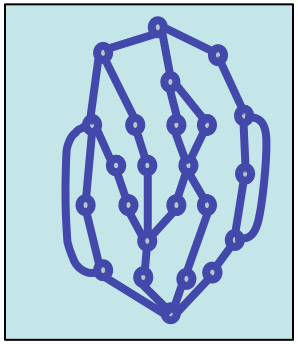
program / model

```
void add(Object o) {
 buffer[head] = o;
 head = (head+1)%size;
}

Object take() {
 …
 tail=(tail+1)%size;
 return buffer[tail];
}
```

property

always(ϕ **or** ψ)

model checking

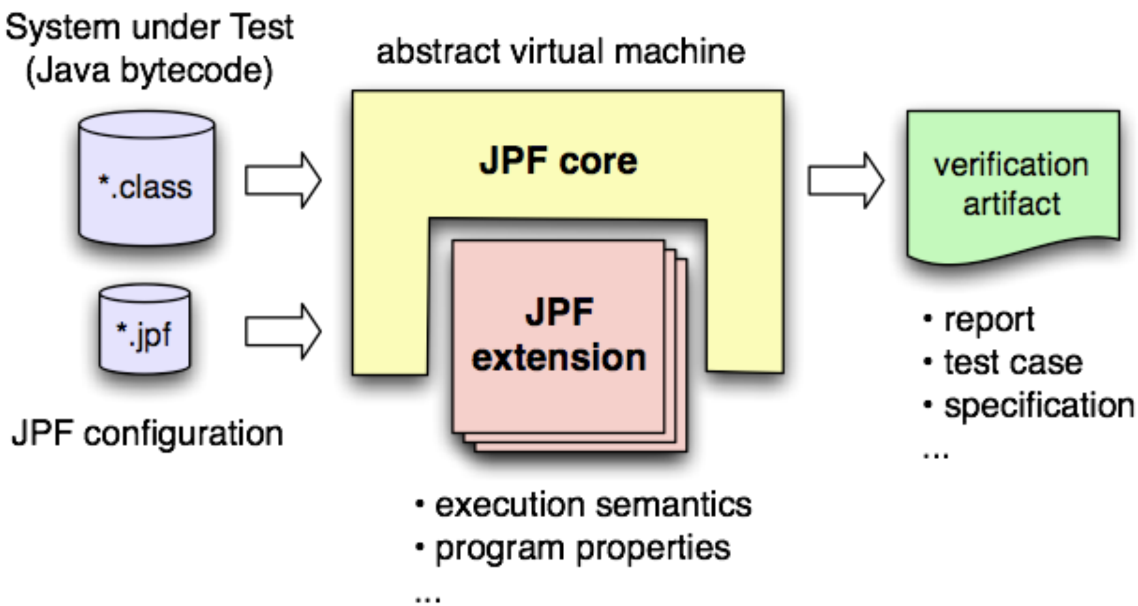


OK

error trace

```
Line 5: …
Line 12: …
…
Line 41:…
Line 47:…
```

Java PathFinder (JPF)



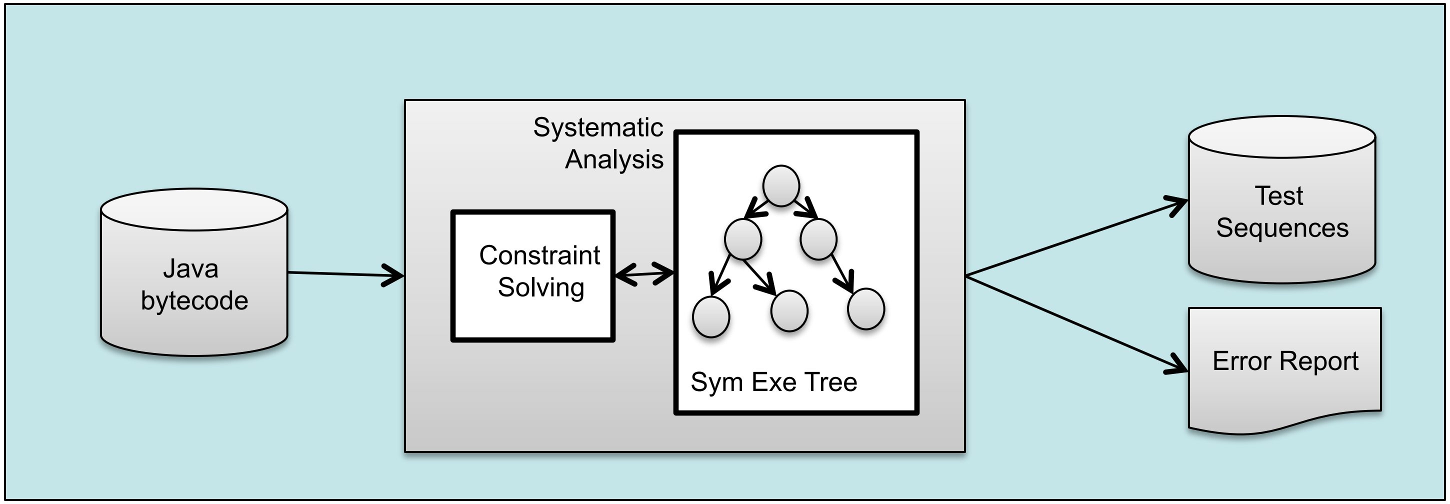
- Extensible virtual machine framework for Java bytecode verification:
- Workbench to implement all kinds of verification tools
- Typical use cases:
  - software model checking (detection of deadlocks, races, assert errors)
  - test case generation (symbolic execution)
  - ... and many more

Java PathFinder (JPF)

- JPF uses scalability enhancing mechanisms
  - on-the-fly partial order reduction
  - configurable search strategies
  - user definable heuristics, choice generators
- Recipient of several awards
  - NASA 2003, IBM 2007, FLC 2009
- Open sourced:

  http://babelfish.arc.nasa.gov/trac/jpf
- Largest application:
  - Fujitsu (one million lines of code)

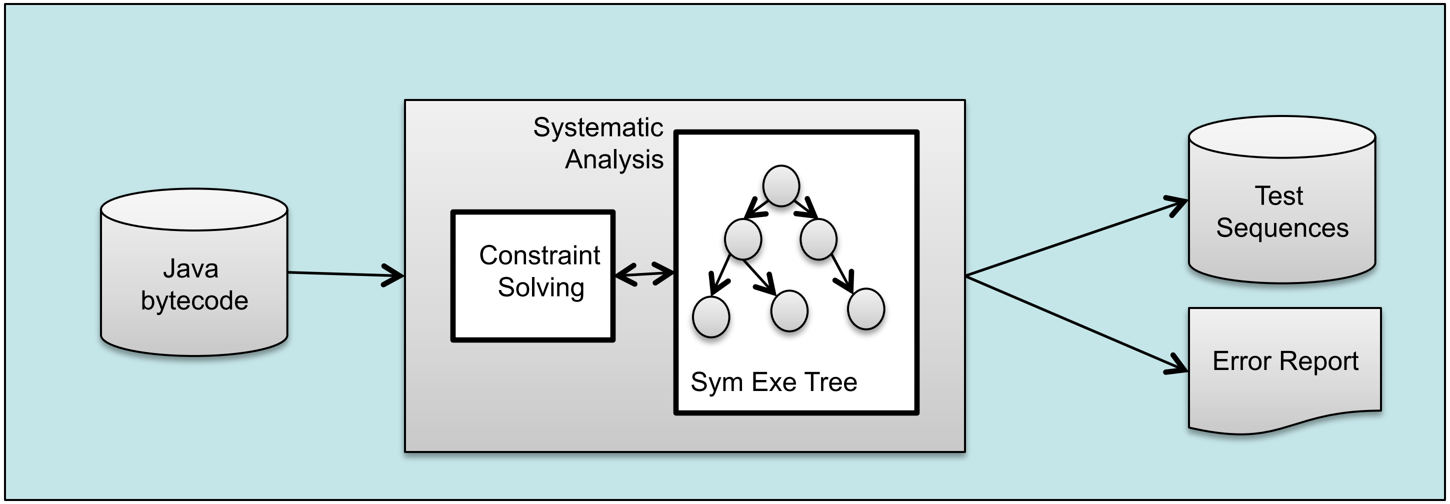
Symbolic PathFinder (SPF)



- Combines symbolic execution, model checking and constraint solving
- Applies to executable models and code
- Handles dynamic data structures, loops, recursion, multi-threading; arrays and strings
- Java PathFinder extension project

[TACAS'03, ISSTA'08, ASE'10]

Symbolic PathFinder (SPF)



Users:

- Academia (uiuc.edu, unl.edu, utexas.edu, byu.edu, umn.edu, Stellenbosch Za, Waterloo Ca, Charles University Prague Cz, …)
- Industry (Fujitsu)
- NASA (Ames, Langley)

Symbolic Execution

Systematic Path Exploration

Generation and Solving of Numeric Constraints

[pres = 460; pres_min = 640; pres_max = 960]

```
if( (pres < pres_min) || (pres > pres_max)) {
        …
} else {
    …
}
```

[pres = Sym_1; pres_min = MIN; pres_max = MAX] [path condition PC: TRUE]

```
if ((pres < pres_min) ||
    (pres > pres_max)) {
      [PC1: Sym1< MIN]
} else {
      …
}
```

```
if ((pres < pres_min)) ||
    (pres > pres_max)) {
      [PC2: Sym1 > MAX]
} else {
      …
}
```

```
if ((pres < pres_min) ||
    (pres > pres_max)) {
      …
} else {
      [PC3: Sym1 >= MIN &&
            Sym1 <= MAX
}
```

Solve path conditions PC_1, PC_2, PC_3 → test inputs

Symbolic Execution

- King [Comm. ACM 1976], Clarke [IEEE TSE 1976]
- Analysis of programs with unspecified inputs
  - Execute a program on symbolic inputs
- Symbolic states represent sets of concrete states
- For each path, build path condition
  - Condition on inputs – for the execution to follow that path
  - Check path condition satisfiability – explore only feasible paths
- Symbolic state
  - Symbolic values/expressions for variables
  - Path condition
  - Program counter

Symbolic Execution

Received renewed interest in recent years

… due to

- Algorithmic advances
- Increased availability of computational power and decision procedures

Applications:

- Test-case generation, error detection, …

Tools, many open-source

- UIUC: CUTE, jCUTE, Stanford: EXE, KLEE, UC Berkeley: CREST, BitBlaze
- Microsoft's Pex, SAGE, YOGI, PREfix
- NASA's Symbolic (Java) Pathfinder
- IBM's Apollo, Parasoft's testing tools etc.

Example – Standard Execution

Code that swaps 2 integers

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Concrete Execution Path

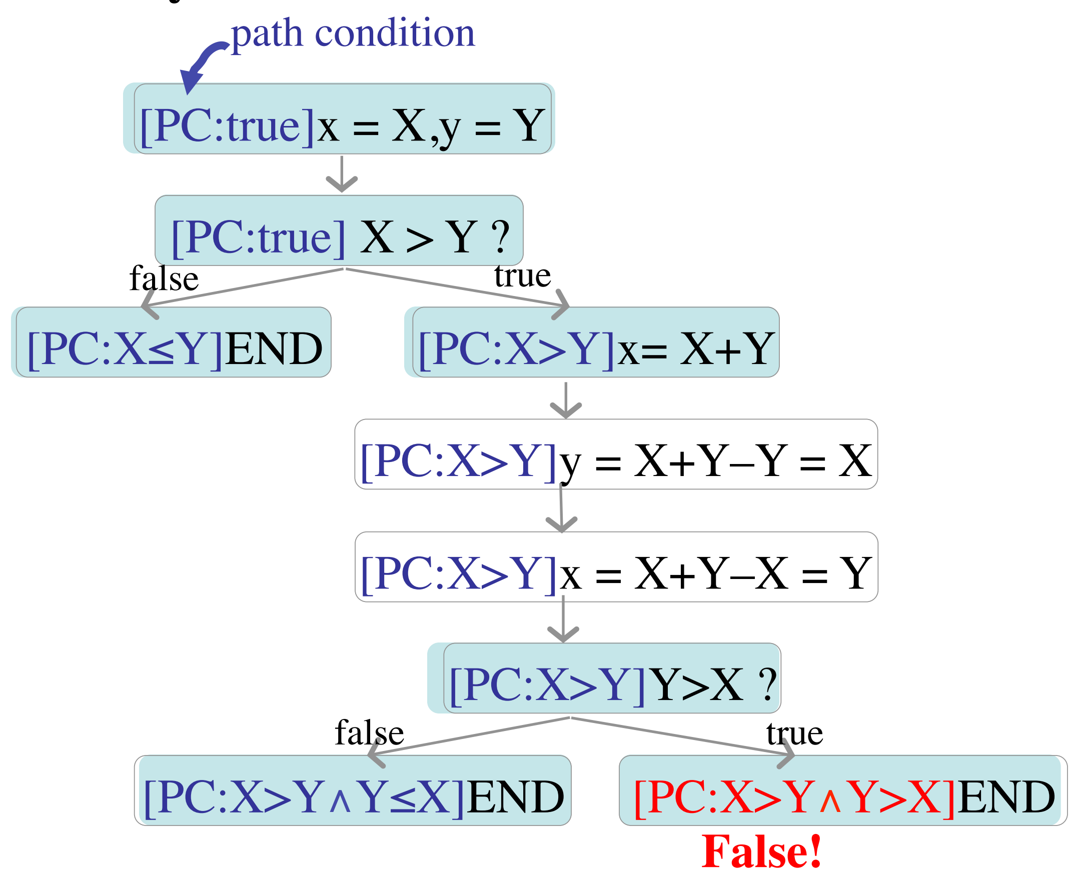
x = 1, y = 0
↓
1 > 0 ? true
↓
x = 1 + 0 = 1
↓
y = 1 – 0 = 1
↓
x = 1 – 1 = 0
↓
0 > 1 ? false

Example – Symbolic Execution

Code that swaps 2 integers

```
int x, y;

if (x > y) {

  x = x + y;

  y = x – y;

  x = x – y;

  if (x > y)

    assert false;

}
```

Symbolic Execution Tree



Solve PCs: obtain test inputs

Symbolic PathFinder (SPF)

Combining Symbolic Execution with Model Checking

Java PathFinder (JPF) used for systematic exploration

- symbolic execution tree
- different heap configurations
  - lazy initialization for input data structures [TACAS' 03]
  - non-determinism handles aliasing in input data structures
- multi-threading
- property checking
- backtracking – when PC un-satisfiable
- different search strategies (depth-first, breadth-first)

Take advantage of JPF's optimizations!

Symbolic PathFinder (SPF)

Combining Symbolic Execution with Model Checking

- No state matching performed
  - Some abstract state matching
- Symbolic search space may be infinite due to loops, recursion
  - We put a limit on the search depth

Implementation

- Non-standard interpreter of byte-codes
  - Replaces concrete execution semantics of byte-codes with symbolic execution
  - Enables JPF-core to perform systematic symbolic analysis
- Attributes
  - Symbolic information stored in attributes associated with the program data
  - Propagated dynamically during symbolic execution

Implementation

- Choice generators
  - handle non-deterministic choices in branching conditions
- Listeners
  - collect and print results: path conditions, test vectors or test sequences
  - influence the search
- Native peers
  - model native libraries
  - e.g. capture `Math` library calls and send them to the constraint solver
- Mixed concrete-symbolic solving

Example: IADD

Concrete execution of IADD byte-code:

```
public class IADD extends
   Instruction { …
 public Instruction execute(…
   ThreadInfo th){
   int v1 = th.pop();
   int v2 = th.pop();
   th.push(v1+v2,…);
   return getNext(th);
 }
}
```

Symbolic execution of IADD byte-code:

```
public class IADD extends
   ….bytecode.IADD { …
 public Instruction execute(…
   ThreadInfo th){
   Expression sym_v1 = ….getOperandAttr(0);
   Expression sym_v2 = ….getOperandAttr(1);
   if (sym_v1 == null && sym_v2 == null)
     // both values are concrete
     return super.execute(… th);
   else {
     int v1 = th.pop();
     int v2 = th.pop();
     th.push(0,…); // don't care
     …
     ….setOperandAttr(Expression._plus(
        sym_v1,sym_v2));
     return getNext(th);
   }
 }
}
```

Example: IFGE

Concrete execution of IFGE byte-code:

```
public class IFGE extends
    Instruction { …
 public Instruction execute(…
    ThreadInfo th){
    cond = (th.pop() >=0);
    if (cond)
     next = getTarget();
    else
     next = getNext(th);
    return next;
 }
}
```

Symbolic execution of IFGE byte-code:

```
public class IFGE extends
   ….bytecode.IFGE { …
 public Instruction execute(…
   ThreadInfo th){
   Expression sym_v = ….getOperandAttr();
   if (sym_v == null)
     // the condition is concrete
     return super.execute(… th);
   else {
     PCChoiceGen cg = new PCChoiceGen(2);…
     cond = cg.getNextChoice()==0?false:true;
     if (cond) {
        pc._add_GE(sym_v,0);
        next = getTarget();
     }
     else {
        pc._add_LT(sym_v,0);
        next = getNext(th);
     }
     if (!pc.satisfiable()) … // JPF backtrack
     else cg.setPC(pc);
     return next;
   } } }
```

Decision Procedures

- Choco, Coral, Yices, CVC3, Hampi, IASolver …
- Generic interface – easy to extend with new constraint solvers and decision procedures

Mathematical functions

Model-level interpretation

`$x + 1` → *Math.sin* → `sin($x + 1)`

Symbolic expression
w/ un-interpreted function handled
directly by solver (Choco)

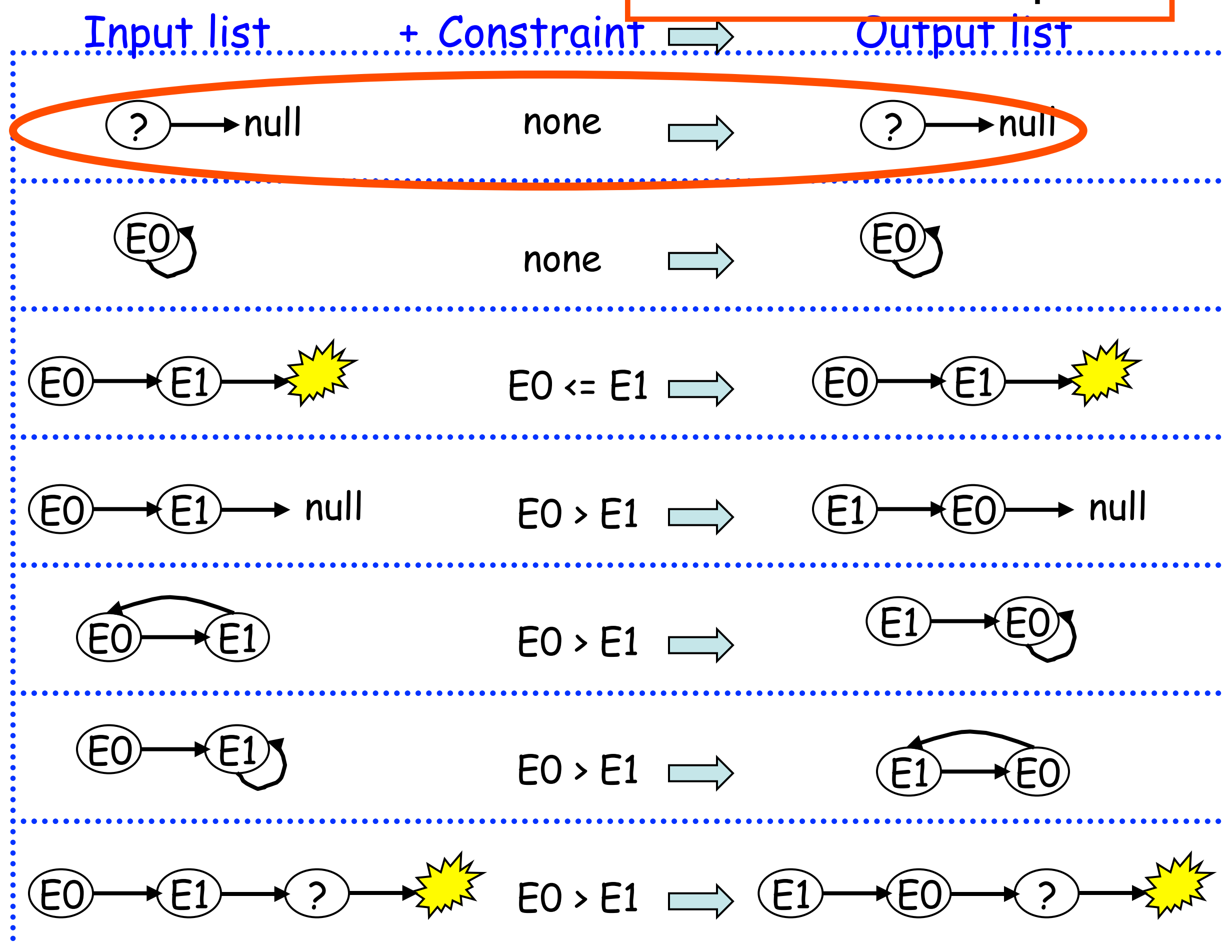
Input Data Structures

- Used to be a challenge
- Lazy initialization [TACAS' 03, SPIN' 05]
- Non-determinism handles aliasing
  - JPF explores different heap configurations explicitly
- Implementation:
  - GETFIELD, GETSTATIC bytecode instructions modified
  - listener prints input heap constraints and method effects (outputs)

Example

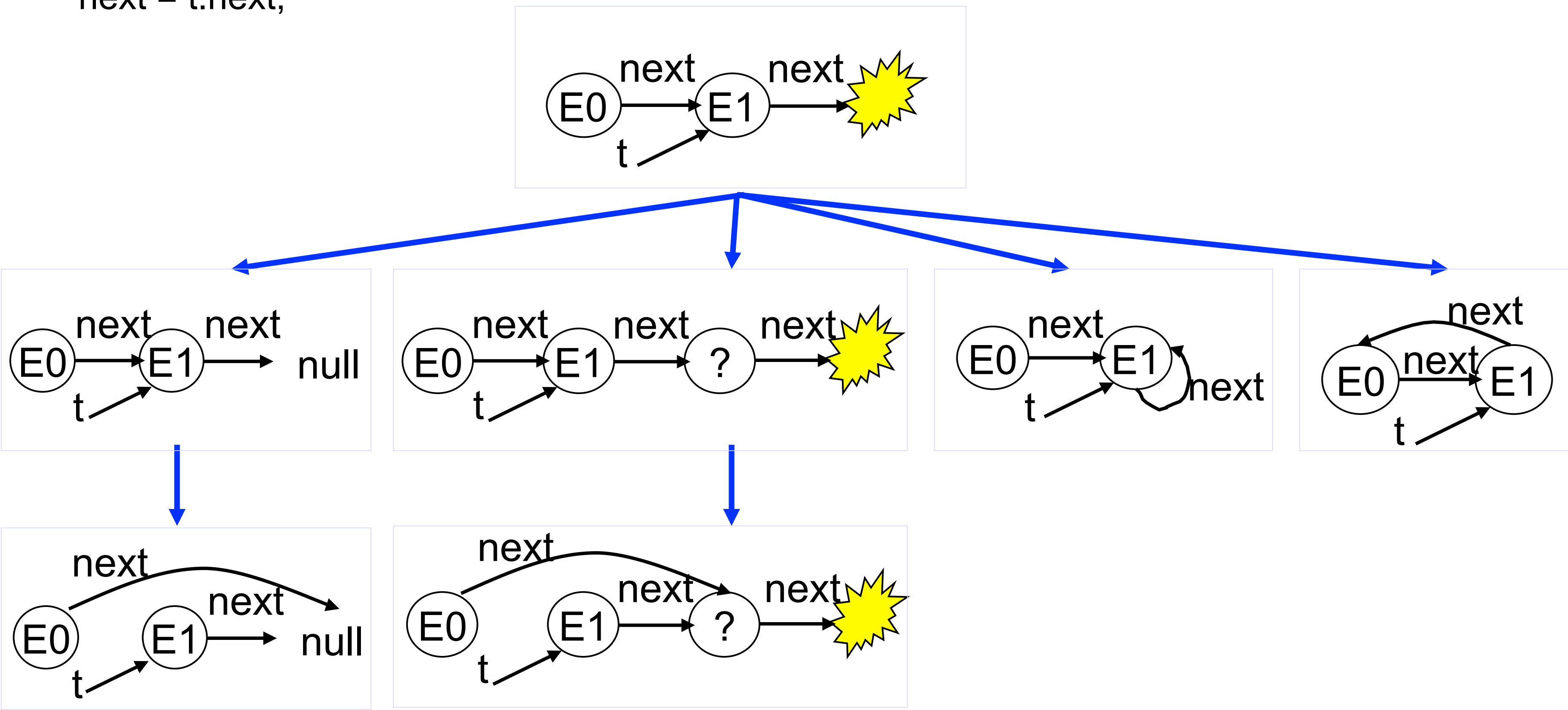
NullPointerException

```
class Node {
    int elem;
    Node next;

    Node swapNode() {
      if (next != null)
        if (elem > next.elem) {
          Node t = next;
          next = t.next;
          t.next = this;
          return t;
        }
      return this;
    }
}
```

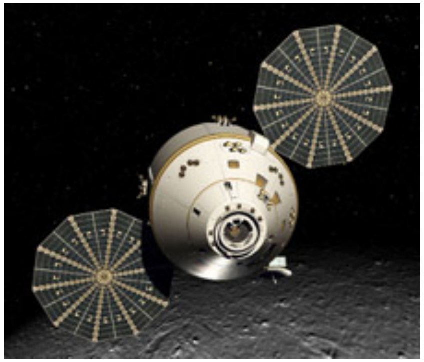


Lazy Initialization

consider executing
next = t.next;



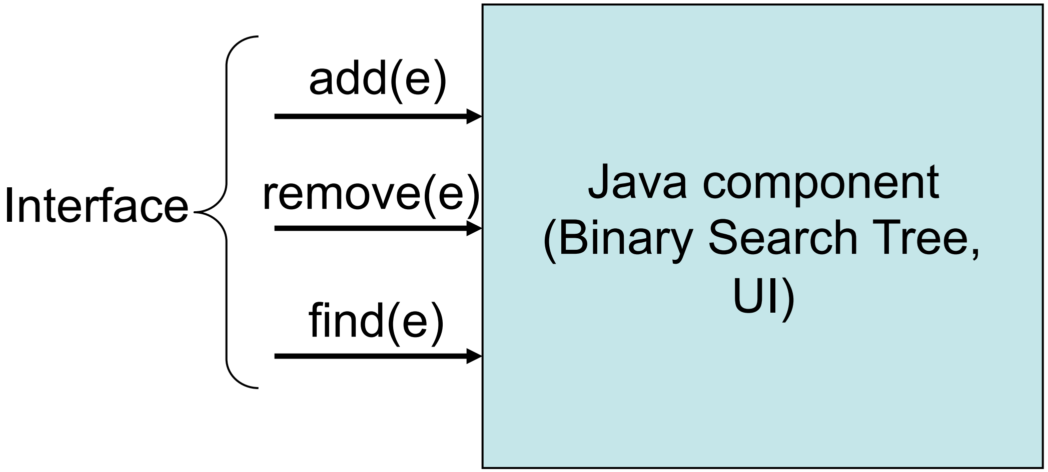
Applications



Orion orbits the moon (Image Credit: Lockheed Martin).

- NASA control software [ISSTA'08]
  - Manual testing: time consuming (~1 week)
  - Guided random testing could not obtain full coverage
  - SPF generated ~200 tests to obtain full coverage in <1min
  - Found major bug in new version
- Polyglot [ISSTA'11, NFM'12]
  - Analysis and test case generation for UML, Stateflow and Rhapsody models
  - Pluggable semantics for different statechart formalisms
  - Analyzed MER Arbiter, Ares-Orion communication
- Tactical Separation Assisted Flight Environment (T-SAFE) [NFM'11, ICST'12]
  - Integration with CORAL for solving complex mathematical constraints
- Test case generation for Android apps
- …

Test Sequence Generation



Generated test sequence:

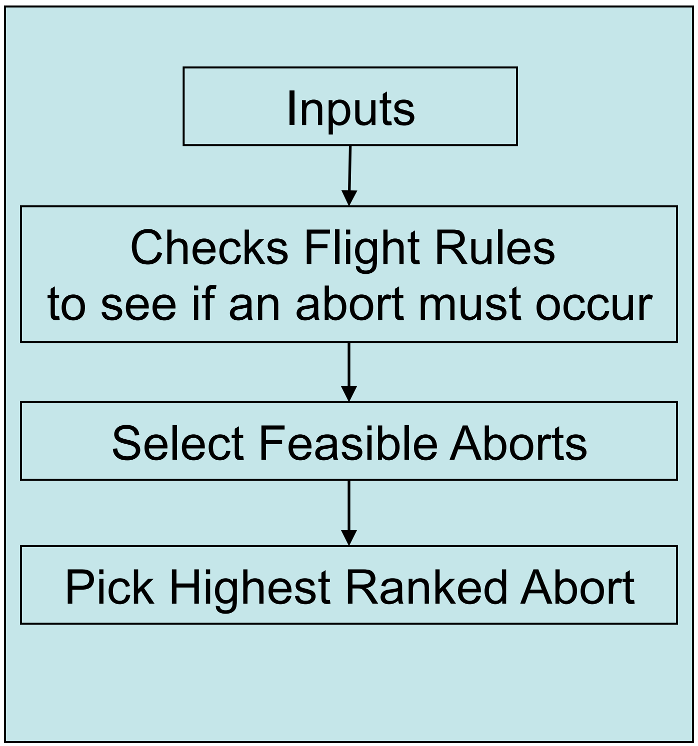
```
BinTree t = new BinTree();
t.add(1);
t.add(2);
t.remove(1);
```

- SymbolicSequenceListener generates JUnit tests:
  - method sequences (up to user-specified depth)
  - method parameters
- JUnit tests can be run directly by the developers
- Measure coverage
- Support for abstract state matching
- Extract specifications

Testing the Onboard Abort Executive (OAE)

Prototype for CEV ascent abort handling being developed by JSC GN&C

OAE Structure



Results

Baseline

- Manual testing: time consuming (~1 week)
- Guided random testing could not cover all aborts

Symbolic PathFinder

- Generates tests to cover all aborts and flight rules
- Total execution time is < 1 min
- Test cases: 151 (some combinations infeasible)
- Errors: 1 (flight rules broken but no abort picked)
- Found major bug in new version of OAE
- Flight Rules: 27 / 27 covered
- Aborts: 7 / 7 covered
- Size of input data: 27 values per test case

[ISSTA'08]

Generated Test Cases and Constraints

Test cases:

// Covers Rule: FR A_2_A_2_B_1: Low Pressure Oxodizer Turbopump speed limit exceeded
// **Output: Abort:IBB**
CaseNum 1;
CaseLine in.stage_speed=3621.0;
CaseTime 57.0-102.0;

// Covers Rule: FR A_2_A_2_A: Fuel injector pressure limit exceeded
// **Output: Abort:IBB**
CaseNum 3;
CaseLine in.stage_pres=4301.0;
CaseTime 57.0-102.0;
…

Constraints:

//Rule: FR A_2_A_1_A: stage1 engine chamber pressure limit exceeded Abort:IA
PC (~60 constraints):
in.geod_alt(9000) < 120000 && in.geod_alt(9000) < 38000 && in.geod_alt(9000) < 10000 &&
in.pres_rate(-2) >= -2 && in.pres_rate(-2) >= -15 &&
in.roll_rate(40) <= 50 && in.yaw_rate(31) <= 41 && in.pitch_rate(70) <= 100 && …

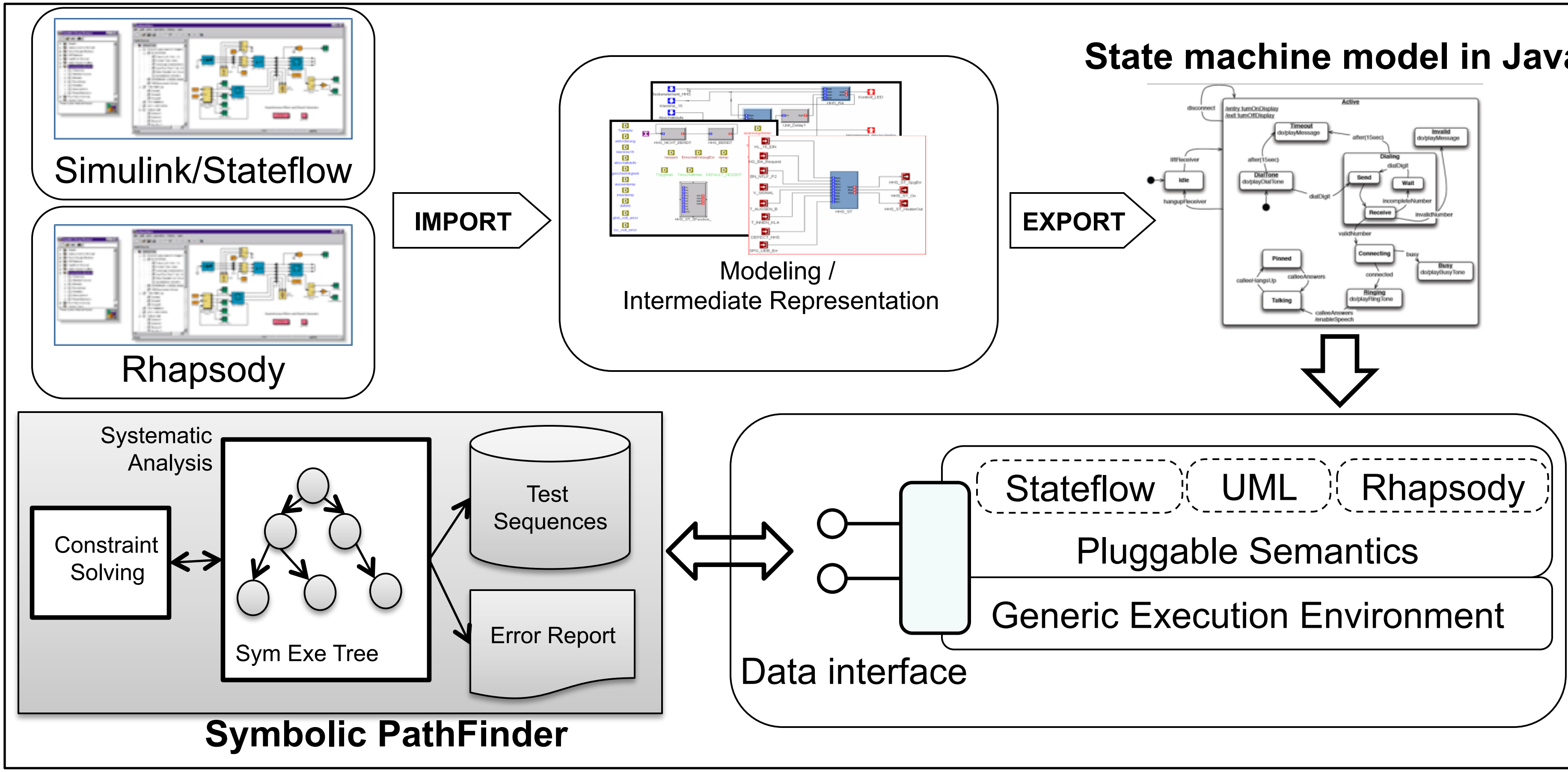
Polyglot

- Large programs such as NASA Exploration
  - Multiple systems that interact via safety-critical protocols
  - Designed with different Statechart variants
  - A unified verification framework needed
- Polyglot
  - Modeling and analysis for multiple Statechart formalisms
  - Captures interactions between components
  - Formal semantics that captures the variants of Statecharts
  - Applied to JPL's MER arbiter, Ares-Orion communication

Collaboration w/ Vanderbilt University and University of Minnesota

[ISSTA'11,NFM'12]

Polyglot



Simulink/Stateflow
Rhapsody
IMPORT
Modeling /
Intermediate Representation
EXPORT
State machine model in Java
Systematic
Analysis
Constraint
Solving
Sym Exe Tree
Test
Sequences
Error Report
Symbolic PathFinder
Stateflow
UML
Rhapsody
Pluggable Semantics
Generic Execution Environment
Data interface

Test Generation for TTEthernet Protocol

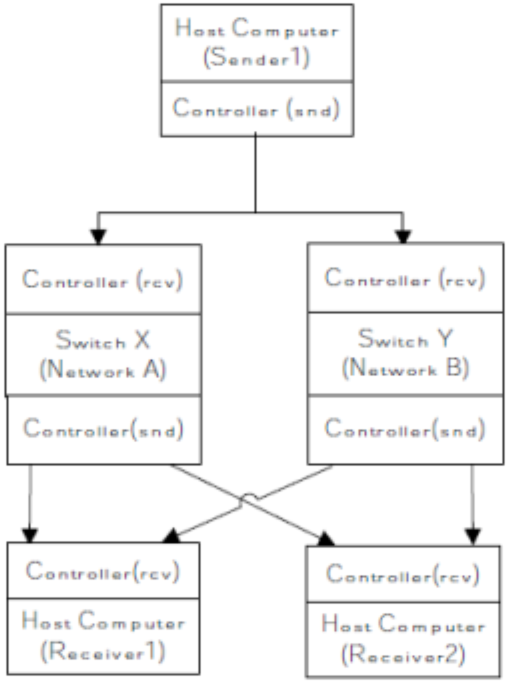
- Fault tolerant version of Ethernet protocol
- Used by NASA in space networks
- Assure reliable network communications.

- Developed PVS model of basic version of the TTEthernet protocol
- Framework for translating models into Java multi-threaded code

SPF analysis

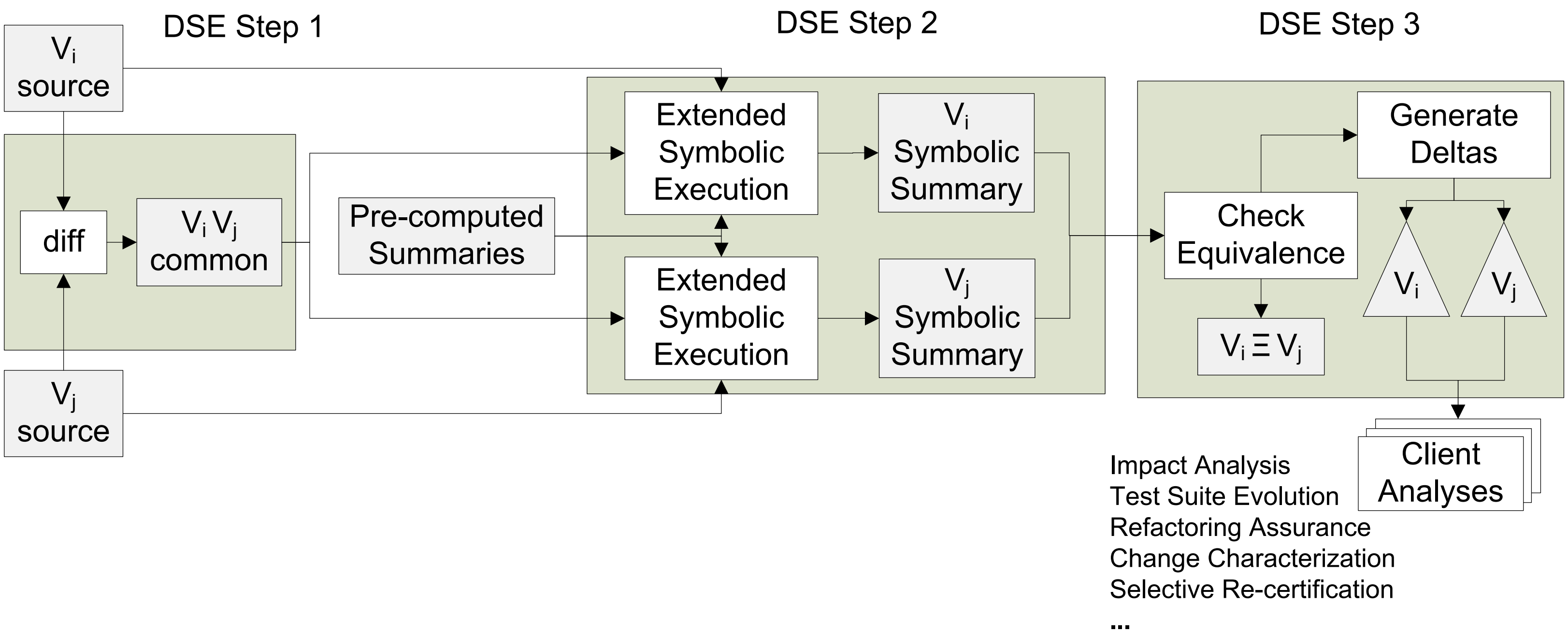
- filtering of test cases to satisfy the various fault hypothesis
- verification of fault-tolerant properties
- demonstrated test case generation for TTEthernet's Single Fault Hypothesis

[w/ NASA Langley]



Shown: Minimal configuration for testing agreement in TTEthernet

Differential Symbolic Execution – NASA Langley



- Computes logical difference between two program versions
- [FSE'08, Person et al PLDI'11]

challenges

RSE

handling complex mathematical constraints

Example constraint generated for a module from TSAFE (Tactical Separation Assisted Flight Environment)

```
sqrt(pow(((x1 + (e1 * (cos(x4) – cos((x4 + (((1.0 * (((c1 * x5) * (e2/c2))/x6)) * x2)/
e1)))))) – (((e2/c2)) * (1.0 – cos((c1 * x5))))),2.0)) > 999.0  & (c1 * x5) > 0.0 &
x3 > 0.0 & x6 > 0.0 & c1 = 0.017… &
c2 = 68443.0 & e1 = ((pow(x2,2.0)/tan((c1*x3)))/c2) &
e2 = pow(x6,2.0)/tan(c1*x3)
```

Coral Solver

- Target application of solver: programs that
  - Use floating-point arithmetic
  - Call math functions

TSAFE example

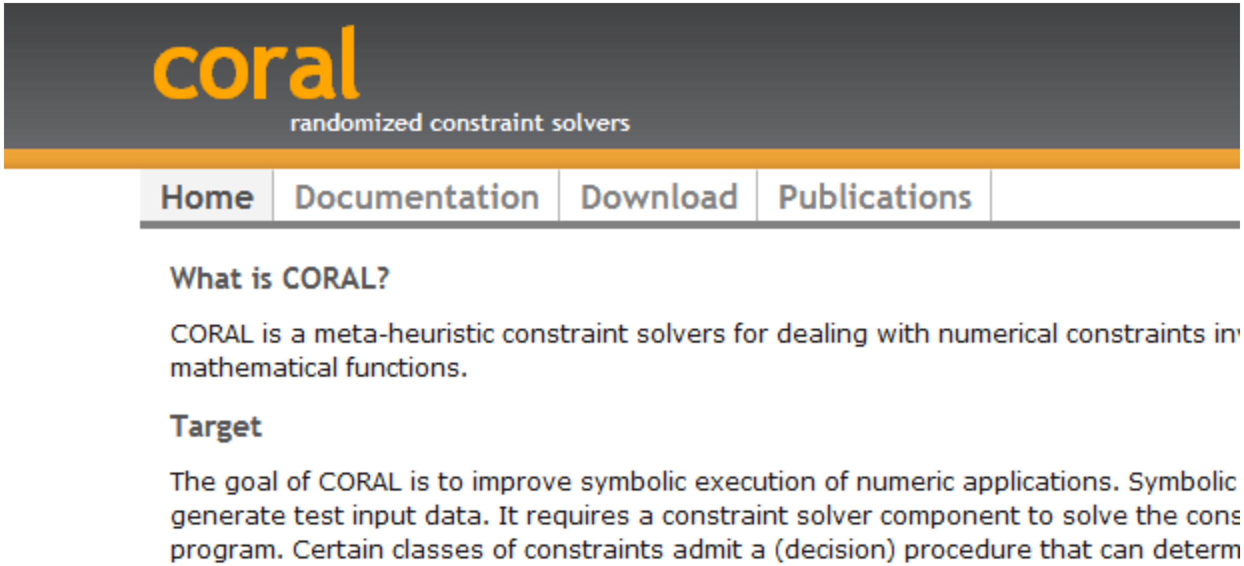
Input: sqrt(pow(((x1 + (e1 * (cos(x4) – …

Output: {x1=100.0, x2=98.48…, x3=3.08…E-11, …}

Approach: combine meta-heuristic search and interval solving [NFM’11, ICST’12]

Coral

http://pan.cin.ufpe.br/coral



coral
randomized constraint solvers

Home | Documentation | Download | Publications

What is CORAL?

CORAL is a meta-heuristic constraint solvers for dealing with numerical constraints in
mathematical functions.

Target

The goal of CORAL is to improve symbolic execution of numeric applications. Symbolic
generate test input data. It requires a constraint solver component to solve the cons
program. Certain classes of constraints admit a (decision) procedure that can determ

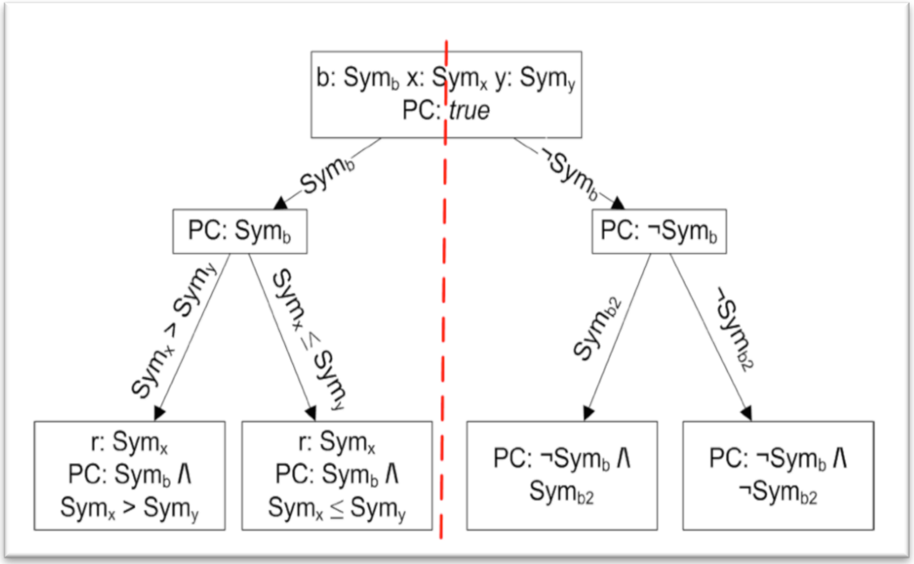
path explosion

parallel symbolic execution [ISSTA'10]

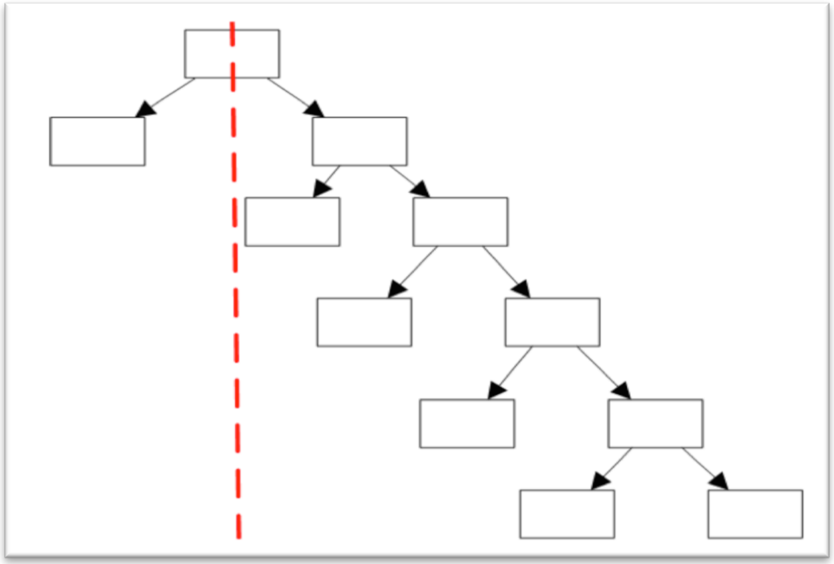
symbolic execution very amenable to parallelization

- no sharing between sub-trees

Balancing partitions



Nicely Balanced – linear speedup



Poorly Balanced – no speedup

- Solutions
  - Simple static partitioning [ISSTA'10]
  - Dynamic partitioning [Andrew King's Masters Thesis at KSU, Cloud9 at EPFL, Fujitsu]

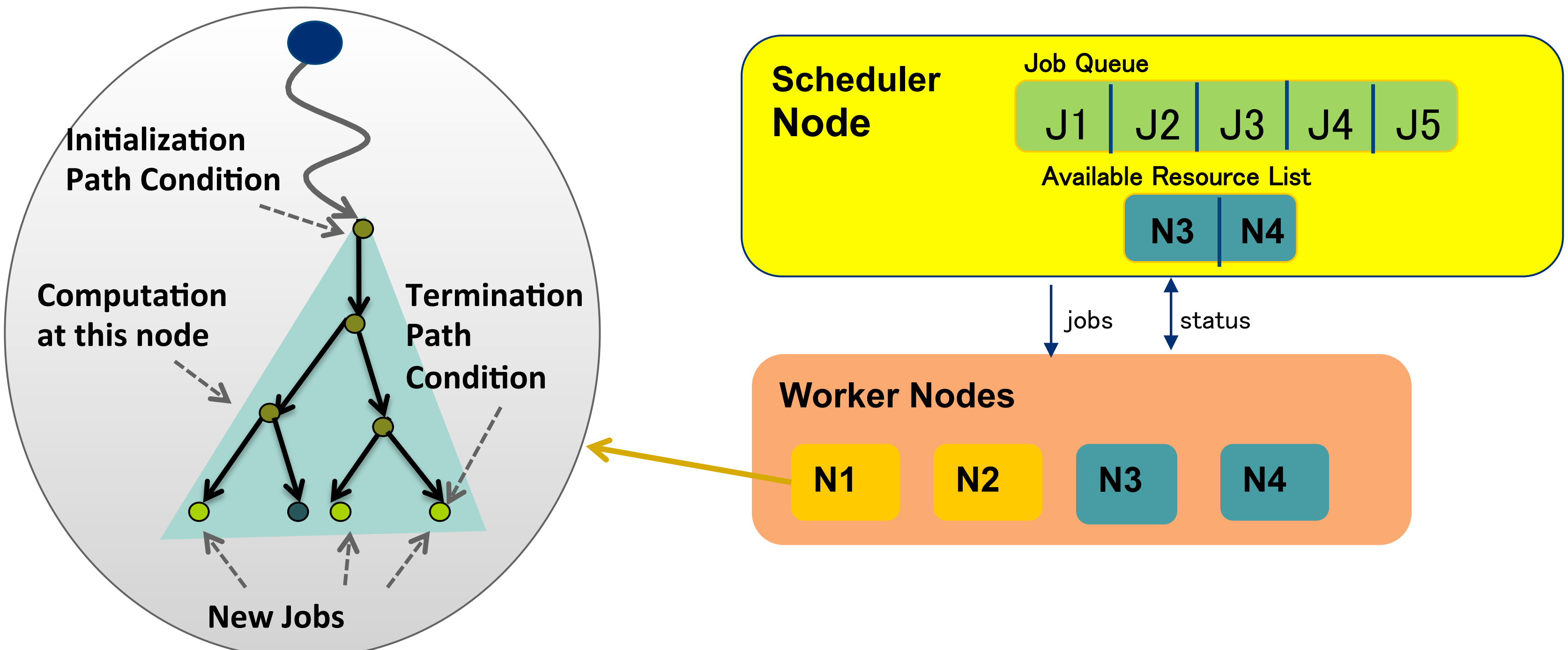
Simple Static Partitioning

- Static partitioning of tree with light dynamic load balancing
  - Flexible, little communication overhead
- Constraint-based partitioning
  - Constraints used as initial pre-conditions
  - Constraints are disjoint and complete
- Approach
  - Shallow symbolic execution => produces large number of constraints
  - Constraints selection – according to frequency of variables
  - Combinatorial partition creation
- Intuition
  - Commonly used variables likely to partition state space in useful ways
- Close to linear speed-up when using 128 workers

Fujitsu applications

Distributed symbolic execution over cloud

- Adaptive dynamic partitioning
- Heuristics to partition jobs on the fly based on system resources and job characteristics and history
- Close to linear speed-up is possible in > 90% of the cases



thanks Fujitsu

Fujitsu applications

Testing web applications – challenge

- Handling complex constraints involving strings and numerics
- e.g.: string s, q; integer a, b;

  s.equals(q) && s.startswith(“uvw”) && q.endswith(“xyz”) && s.length() <a && (a+b)<6 && b>0 Unsatisfiable !

Solution – string solver

- Maintain separate constraint set for Integer/Boolean and Real – represented as equations
- Maintain separate constraint set for string variables – represented as FSMs or regular expressions
- Pass learned constraints from one domain to another and iterate to fixed point or time out

Fujitsu technology handles symbolic execution and test case generation for web applications which uses String input variables extensively

thanks Fujitsu

handling native code

```
void test(int x, int y) {
    if (x > 0) {
      if (y == hash(x))
        S0;
      else
        S1;
      if (x > 3 && y > 10)
        S3;
      else
        S4;
    }
}
```

S0, S1, S3, S4 = statements we wish to cover

hash is native or can not be handled by decision procedure

handling native code

```
void test(int x, int y) {
  if (x > 0) {
   if (y == hash(x))
     S0;
   else
     S1;
   if (x > 3 && y > 10)
     S3;
   else
     S4;
  }
}
```

hash is native or can not be handled by decision procedure

S0, S1, S3, S4 = statements we wish to cover

Symbolic Execution
Can not handle it!

Solution:
Mixed concrete-symbolic solving [ISSTA'11]

Mixed Concrete-Symbolic Solving

- Use un-interpreted functions for external library calls
- Split path condition PC into:
  - simplePC – solvable constraints
  - complexPC – non-linear constraints with un-interpreted functions
- Solve simplePC
  - Use obtained solutions to simplify complexPC
- Check the result again for satisfiability

Mixed Concrete-Symbolic Solving

Assume hash(x) = 10 *x:

PC: $\underbrace{X>3 \ \wedge \ Y>10}_{\text{simplePC}} \ \wedge \ \underbrace{Y=hash(X)}_{\text{complexPC}}$

Solve simplePC

Use solution X=4 to compute h(4)=40

Simplify complexPC: Y=40

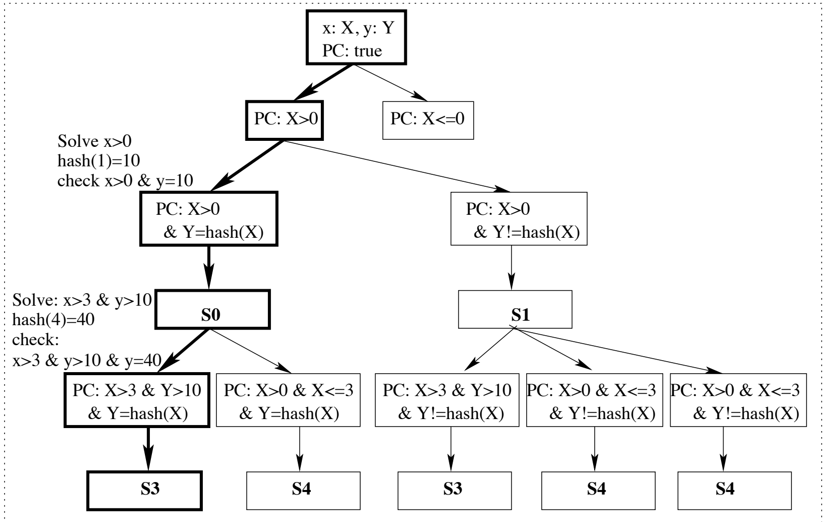
Solve again:

simplified PC: X>3 ∧ Y>10 ∧ Y=40 Satisfiable!

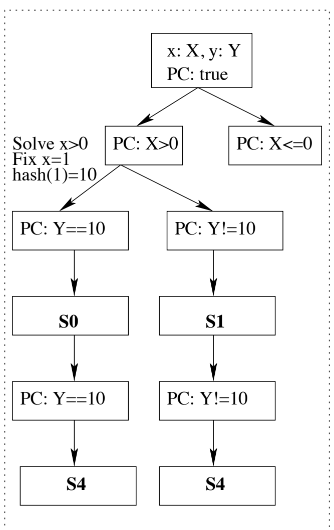
example

```
void test(int x, int y) {
  if (x > 0) {
   if (y == hash(x))  //hash(x)=10*x
     S0;
   else
     S1;
   if (x > 3 && y > 10)
   //if (y > 10)
     S3;
   else
     S4;
  }
}
```

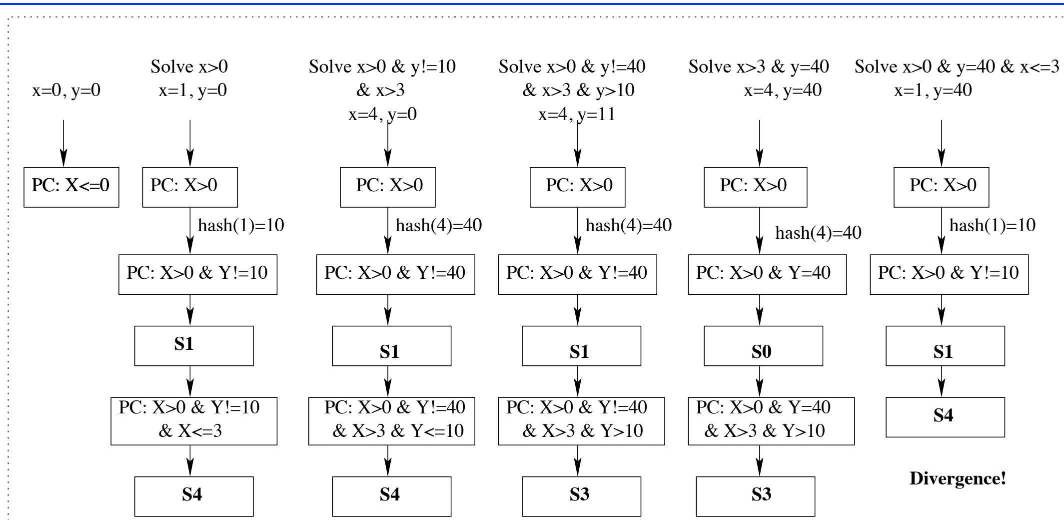
Example



Mixed concrete-symbolic solving: all paths covered



EXE results: stmt “S3” not covered



Predicted path “S0;S4”
!= path taken “S1;S4”

DART results: path “S0;S4” not covered

Mixed Concrete-Symbolic Solving vs DART

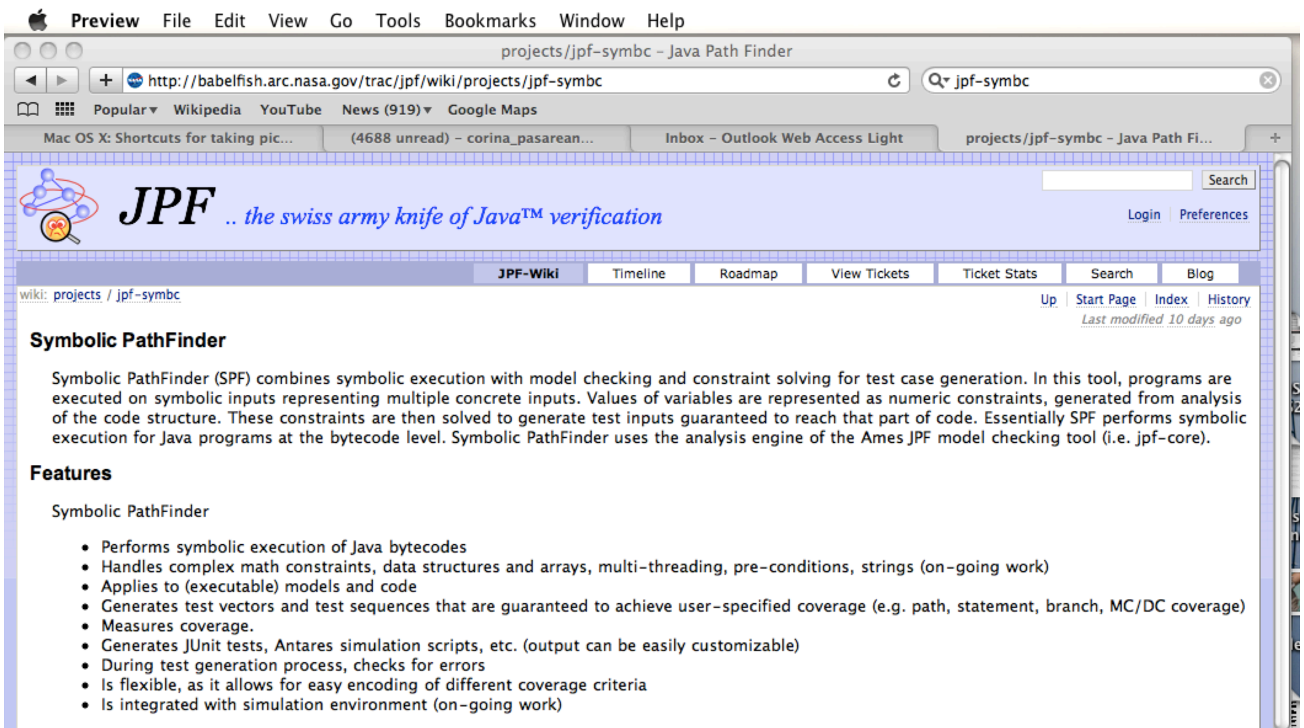
- DART = Directed Automated Random Testing
  - Collects symbolic constraints during concrete executions
- Both techniques incomplete
- Incomparable in power (see paper)
- Mixed concrete-symbolic solving can handle only “pure”, side-effect free functions
  - DART does not have the limitation; will likely diverge

Symbolic PathFinder

Combining Symbolic Execution with Model Checking

Available from JPF distribution:

http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc


Preview File Edit View Go Tools Bookmarks Window Help

projects/jpf-symbc – Java Path Finder

http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc

jpf-symbc

Popular Wikipedia YouTube News (919) Google Maps

Mac OS X: Shortcuts for taking pic... | (4688 unread) – corina_pasarean... | Inbox – Outlook Web Access Light | projects/jpf-symbc – Java Path Fi...

JPF .. the swiss army knife of Java™ verification

Search

Login | Preferences

JPF-Wiki | Timeline | Roadmap | View Tickets | Ticket Stats | Search | Blog

wiki: projects / jpf-symbc

Up | Start Page | Index | History

Last modified 10 days ago

Symbolic PathFinder

Symbolic PathFinder (SPF) combines symbolic execution with model checking and constraint solving for test case generation. In this tool, programs are executed on symbolic inputs representing multiple concrete inputs. Values of variables are represented as numeric constraints, generated from analysis of the code structure. These constraints are then solved to generate test inputs guaranteed to reach that part of code. Essentially SPF performs symbolic execution for Java programs at the bytecode level. Symbolic PathFinder uses the analysis engine of the Ames JPF model checking tool (i.e. jpf-core).

Features

Symbolic PathFinder

- Performs symbolic execution of Java bytecodes
- Handles complex math constraints, data structures and arrays, multi-threading, pre-conditions, strings (on-going work)
- Applies to (executable) models and code
- Generates test vectors and test sequences that are guaranteed to achieve user-specified coverage (e.g. path, statement, branch, MC/DC coverage)
- Measures coverage.
- Generates JUnit tests, Antares simulation scripts, etc. (output can be easily customizable)
- During test generation process, checks for errors
- Is flexible, as it allows for easy encoding of different coverage criteria
- Is integrated with simulation environment (on-going work)

RSE

Related Approaches

- Korat: black box test generation [Boyapati et al. ISSTA' 02]
- Concolic execution [Godefroid et al. PLDI' 05, Sen et al. ESEC/FSE' 05]
  - DART/CUTE/jCUTE/…
- Concrete model checking with abstract matching and refinement [CAV' 05]
- Symstra [Xie et al. TACAS' 05]
- Execution Generated Test Cases [Cadar & Engler SPIN' 05]
- Testing, abstraction, theorem proving: better together! [Yorsh et al. ISSTA' 06]
- SYNERGY: a new algorithm for property checking [Gulavi et al. FSE' 06]
- Feedback directed random testing [Pacheco et al. ICSE' 07]
- …

Current and Future Work

- Symbolic execution for program specialization
- Thread-modular reasoning
- Memoization across multiple SPF runs [ISSTA'12]
- Testing for Android applications
- Reliability analysis – compute probability of reaching a fault state
- Invariant generation [SPIN'04]
- Program and test repair

RSE

Thank you!